Agenda (in total 7 Sessions)

- **Session 1: OpenMP Introduction**
  - Welcome
  - OpenMP Overview
  - Parallel Region
  - Worksharing
  - Scoping
  - Tasking (short introduction)
  - Executing OpenMP programs
  - Homework assignments 😊
  - Compile and run on Perlmutter CPUs

- **Session 2: Tasking**
- **Session 3: Optimization for NUMA and SIMD**
- **Session 4: What Could Possibly Go Wrong Using OpenMP**
- **Session 5: Introduction to Offloading with OpenMP**
- **Session 6: Advanced OpenMP Offloading Topics**
- **Session 7: Selected / Remaining Topics**
Programming OpenMP

An Overview Of OpenMP

Christian Terboven
Michael Klemm
History

- De-facto standard for Shared-Memory Parallelization.
- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN
- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 now includes both programming languages.
- 05/2008: OpenMP 3.0
- 07/2011: OpenMP 3.1
- 07/2013: OpenMP 4.0
- 11/2015: OpenMP 4.5
- 11/2018: OpenMP 5.0
- 11/2020: OpenMP 5.1
- 11/2021: OpenMP 5.2

RWTH Aachen University is a member of the OpenMP Architecture Review Board (ARB) since 2006. Main topics:
- Affinity
- Tasking
- Tool support
- Accelerator support
What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- Memory Management
- ...
What is OpenMP?

- Parallel Region & Worksharing
- Tasking
- SIMD / Vectorization
- Accelerator Programming
- Memory Management
- ...

Parallel Region

Worksharing

Tasking

SIMD / Vectorization

Accelerator Programming

Memory Management

Vectorization

Accelerators
Recent Books About OpenMP

- A printed copy of the 5.2 specifications, 2021
- A book that covers all of the OpenMP 4.5 features, 2017
- A book about the OpenMP Common Core, 2019
Programming OpenMP

Parallel Region

Christian Terboven
Michael Klemm
OpenMP’s machine model

- OpenMP: Shared-Memory Parallel Programming Model.

All processors/cores access a shared main memory.

Real architectures are more complex, as we will see later / as we

Parallelization in OpenMP employs multiple threads.
The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s) in private memory
- Data transfer is through shared memory and is 100% transparent to the application
The OpenMP Execution Model

- OpenMP programs start with just one thread: The *Initial Thread*.

- *Worker* threads are spawned at *Parallel Regions*, together with the initial thread they form the *Team* of threads.

- In between Parallel Regions the Worker threads are put to sleep. The OpenMP *Runtime* takes care of all thread management work.

- Concept: *Fork-Join*.
- Allows for an incremental parallelization!
Parallel Region and Structured Blocks

- The parallelism has to be expressed explicitly.

```
C/C++
#pragma omp parallel
{
  ...
  structured block
  ...
}
```

- Fortran
```
!$omp parallel
  ...
  structured block
  ...
!$omp end parallel
```

- **Structured Block**
  - Exactly one entry point at the top
  - Exactly one exit point at the bottom
  - Branching in or out is not allowed
  - Terminating the program is allowed (abort / exit)

- **Specification of number of threads:**
  - Environment variable: `OMP_NUM_THREADS=...`
  - Or: Via `num_threads` clause:
    `add num_threads(num) to the parallel construct`
Programming OpenMP

Using OpenMP Compilers

Christian Terboven
Michael Klemm
Production Compilers w/ OpenMP Support

- GCC
- clang/LLVM
- HPE CPE
- AOCC, AOMP, ROCmCC
- Intel Classic and Next-gen Compilers
- IBM XL
- … and many more

- See https://www.openmp.org/resources/openmp-compilers-tools/ for a list
Compiling OpenMP

- Enable OpenMP via the compiler’s command-line switches
  - GCC: -fopenmp
  - clang: -fopenmp
  - HPE/Cray CPE: -homp or -fopenmp
  - AOCC, AOCL, ROCmCC: -fopenmp
  - Intel: -fopenmp or -qopenmp (classic) or -fiopenmp (next-gen)
  - IBM XL: -qsmp=omp

- Switches have to be passed to both compiler and linker:

```
$ gcc [...] -fopenmp -o matmul.o -c matmul.c
$ gcc [...] -fopenmp -o matmul matmul.o
$./matmul 1024
Sum of matrix (serial):  134217728.000000, wall time 0.413975, speed-up 1.00
Sum of matrix (parallel): 134217728.000000, wall time 0.092162, speed-up 4.49
```
Starting OpenMP Programs on Linux

• From within a shell, global setting of the number of threads:
  
  ```sh
  export OMP_NUM_THREADS=4
  ./program
  ```

• From within a shell, one-time setting of the number of threads:

  ```sh
  OMP_NUM_THREADS=4   ./program
  ```
Hello OpenMP World
Programming OpenMP

Worksharing

Christian Terboven
Michael Klemm
For Worksharing

- If only the `parallel` construct is used, each thread executes the Structured Block.
- Program Speedup: Worksharing
- OpenMP’s most common Worksharing construct: `for`

```c
int i;
#pragma omp for
for (i = 0; i < 100; i++)
{
    a[i] = b[i] + c[i];
}
```

```fortran
INTEGER :: i
!$omp do
DO i = 0, 99
    a[i] = b[i] + c[i]
END DO
```

- Distribution of loop iterations over all threads in a Team.
- Scheduling of the distribution can be influenced.

- Loops often account for most of a program’s runtime!
Worksharing illustrated

Pseudo-Code
Here: 4 Threads

Thread 1
do i = 0, 24
  a(i) = b(i) + c(i)
end do

Thread 2
do i = 25, 49
  a(i) = b(i) + c(i)
end do

Thread 3
do i = 50, 74
  a(i) = b(i) + c(i)
end do

Thread 4
do i = 75, 99
  a(i) = b(i) + c(i)
end do

Serial
do i = 0, 99
  a(i) = b(i) + c(i)
end do

Memory
A(0)
A(99)
B(0)
B(99)
C(0)
C(99)
The Barrier Construct

- **OpenMP barrier** (implicit or explicit)
  - Threads wait until all threads of the current *Team* have reached the barrier
    
    ```c/c++
    #pragma omp barrier
    ```

- All worksharing constructs contain an implicit barrier at the end
The Single Construct

- The **single** construct specifies that the enclosed structured block is executed by only one thread of the team.
  - It is up to the runtime which thread that is.

- Useful for:
  - I/O
  - Memory allocation and deallocation, etc. (in general: setup work)
  - Implementation of the single-creator parallel-executor pattern as we will see later...

### C/C++

```c
#pragma omp single [clause]
... structured block ...
```

### Fortran

```fortran
!$omp single [clause]
... structured block ...
!$omp end single
```
The Master Construct (will be deprecated in OpenMP 6.0)

- The `master` construct specifies that the enclosed structured block is executed only by the master thread of a team.
  - Replacement: see the `masked` construct later

- Note: The masked construct is no worksharing construct and does not contain an implicit barrier at the end.

**C/C++**

```c
c#pragma omp master[clause]
... structured block ...
```

**Fortran**

```fortran
!$omp master[clause]
... structured block ...
!$omp end master
```
Vector Addition
Influencing the For Loop Scheduling

- **for-construct**: OpenMP allows to influence how the iterations are scheduled among the threads of the team, via the `schedule` clause:

  - `schedule(static [, chunk])`: Iteration space divided into blocks of chunk size, blocks are assigned to threads in a round-robin fashion. If chunk is not specified: #threads blocks.

  - `schedule(dynamic [, chunk])`: Iteration space divided into blocks of chunk (not specified: 1) size, blocks are scheduled to threads in the order in which threads finish previous blocks.

  - `schedule(guided [, chunk])`: Similar to dynamic, but block size starts with implementation-defined value, then is decreased exponentially down to chunk.

- **Default** is `schedule(static)`. 
Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - `schedule(static [, chunk])`
  - Decomposition depending on chunksize
  - Equal parts of size ‘chunksize’ distributed in round-robin fashion
Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - `schedule(static [, chunk])`
  - Decomposition depending on chunksize
  - Equal parts of size ‘chunksize’ distributed in round-robin fashion
- **Pros?**
Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - `schedule(static [, chunk])`
  - Decomposition depending on chunksize
  - Equal parts of size ‘chunksize’ distributed in round-robin fashion

- **Pros?**
  - No/low runtime overhead
Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - `schedule(static [, chunk])`
  - Decomposition depending on chunksize
  - Equal parts of size ‘chunksize’ distributed in round-robin fashion

- **Pros?**
  - No/low runtime overhead

- **Cons?**
Influencing the For Loop Scheduling / 2

- **Static Schedule**
  - `schedule(static [, chunk])`
  - Decomposition depending on chunksize
  - Equal parts of size ‘chunksize’ distributed in round-robin fashion

- **Pros?**
  - No/low runtime overhead

- **Cons?**
  - No dynamic workload balancing
Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1
• Dynamic schedule
  – `schedule(dynamic [, chunk])`
  – Iteration space divided into blocks of chunk size
  – Threads request a new block after finishing the previous one
  – Default chunk size is 1

• Pros ?
• Dynamic schedule
  – `schedule(dynamic [, chunk])`
  – Iteration space divided into blocks of chunk size
  – Threads request a new block after finishing the previous one
  – Default chunk size is 1

• Pros ?
  – Workload distribution
Influencing the For Loop Scheduling / 3

- **Dynamic schedule**
  - `schedule(dynamic [, chunk])`
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1

- **Pros ?**
  - Workload distribution

- **Cons?**
Influencing the For Loop Scheduling / 3

- Dynamic schedule
  - schedule(dynamic [, chunk])
  - Iteration space divided into blocks of chunk size
  - Threads request a new block after finishing the previous one
  - Default chunk size is 1

- Pros?
  - Workload distribution

- Cons?
  - Runtime Overhead
  - Chunk size essential for performance
  - No NUMA optimizations possible
Synchronization Overview

• Can all loops be parallelized with `for`-constructs? No!
  – Simple test: If the results differ when the code is executed backwards, the loop iterations are not independent. BUT: This test alone is not sufficient:

```
C/C++
int i, int s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
   s = s + a[i];
}
```

• Data Race: If between two synchronization points at least one thread writes to a memory location from which at least one other thread reads, the result is not deterministic (race condition).
Synchronization: Critical Region

- A **Critical Region** is executed by all threads, but by only one thread simultaneously (**Mutual Exclusion**).

```c/c++
#include <omp.h>

int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    { s = s + a[i]; }
}
```

- Do you think this solution scales well?

```c/c++
#include <omp.h>

int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    { s = s + a[i]; }
}
```
Programming OpenMP

Scoping

Christian Terboven
Michael Klemm
Scoping Rules

- Managing the Data Environment is the challenge of OpenMP.

- **Scoping** in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    - *firstprivate*: Initialization with the value before encountering the construct
    - *lastprivate*: Value of last loop iteration is written back to the initial thread
  - Static variables are *shared*
Scoping Rules

• Managing the Data Environment is the challenge of OpenMP.

• *Scoping* in OpenMP: Dividing variables in *shared* and *private*:
  - *private*-list and *shared*-list on Parallel Region
  - *private*-list and *shared*-list on Worksharing constructs
  - General default is *shared* for Parallel Region, *firstprivate* for Tasks.
  - Loop control variables on *for*-constructs are *private*
  - Non-static variables local to Parallel Regions are *private*
  - *private*: A new uninitialized instance is created for the task or each thread executing the construct
    • *firstprivate*: Initialization with the value before encountering the construct
    • *lastprivate*: Value of last loop iteration is written back to the initial thread
  - Static variables are *shared*
Privatization of Global/Static Variables

- Global / static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword __thread (GNU extension)

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int i;</td>
<td>SAVE INTEGER :: i</td>
</tr>
<tr>
<td>#pragma omp threadprivate(i)</td>
<td>!$omp threadprivate(i)</td>
</tr>
</tbody>
</table>
Privatization of Global/Static Variables

- Global/static variables can be privatized with the *threadprivate* directive
  - One instance is created for each thread
    - Before the first parallel region is encountered
    - Instance exists until the program ends
    - Does not work (well) with nested Parallel Region
  - Based on thread-local storage (TLS)
    - TlsAlloc (Win32-Threads), pthread_key_create (Posix-Threads), keyword __thread (GNU extension)

Really: try to avoid the use of threadprivate and static variables!

<table>
<thead>
<tr>
<th>C/C++</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>static int i;</td>
<td>SAVE INTEGER :: i</td>
</tr>
<tr>
<td>#pragma omp threadprivate(i)</td>
<td>!$omp threadprivate(i)</td>
</tr>
</tbody>
</table>
Back to our example

C/C++

```c
int i, s = 0;
#pragma omp parallel for
for (i = 0; i < 100; i++)
{
    #pragma omp critical
    {
        s = s + a[i];
    }
}
```
It's your turn: Make It Scale!

```c
#pragma omp parallel
{

#pragma omp for
for (i = 0; i < 99; i++)
{
    s = s + a[i];
}

} // end parallel
```

```c
do i = 0, 24
    s = s + a(i)
end do

do i = 25, 49
    s = s + a(i)
end do

do i = 50, 74
    s = s + a(i)
end do

do i = 75, 99
    s = s + a(i)
end do
```
```c
#pragma omp parallel
{
    double ps = 0.0; // private variable
#pragma omp for
    for (i = 0; i < 99; i++)
    {
        ps = ps + a[i];
    }
#pragma omp critical
    {
        s += ps;
    }
} // end parallel
```

```c
do i = 0, 99
    s = s + a(i)
end do
```
The Reduction Clause

- In a *reduction*-operation the operator is applied to all variables in the list. The variables have to be *shared*.
  - reduction(operator:list)
  - The result is provided in the associated reduction variable

```
C/C++
int i, s = 0;
#pragma omp parallel for reduction(+:s)
for(i = 0; i < 99; i++)
{
    s = s + a[i];
}
```

- Possible reduction operators with initialization value:
  + (0), * (1), – (0), & (~0), | (0), && (1), || (0), ^ (0), min (largest number), max (least number)
- Remark: OpenMP also supports user-defined reductions (not covered here)
Example

\[ PI \]
Example: Pi (1/2)

```c
double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH   = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
```

\[ \pi = \int_0^1 \frac{4}{1 + x^2} \]
Example: Pi (2/2)

double f(double x)
{
    return (4.0 / (1.0 + x*x));
}

double CalcPi (int n)
{
    const double fH = 1.0 / (double) n;
    double fSum = 0.0;
    double fX;
    int i;

    #pragma omp parallel for private(fX,i) reduction(+:fSum)
    for (i = 0; i < n; i++)
    {
        fX = fH * ((double)i + 0.5);
        fSum += f(fX);
    }
    return fH * fSum;
}
Programming OpenMP

OpenMP Tasking Introduction

Christian Terboven
Michael Klemm
Tasking Execution Model

- Supports unstructured parallelism
  - unbounded loops
  - recursive functions

- Several scenarios are possible:
  - single creator, multiple creators, nested tasks (tasks & WS)
  - All threads in the team are candidates to execute tasks

Example (unstructured parallelism)

```c
#pragma omp parallel
#pragma omp masked
while (elem != NULL) {
    #pragma omp task
    compute(elem);
    elem = elem->next;
}
```

```
while ( <expr> ) {
    ...
}
```

```c
void myfunc( <args> )
{
    ...; myfunc( <newargs> ); ...;
}
```
What is a Task in OpenMP?

- Tasks are work units whose execution
  → may be deferred or…
  → … can be executed immediately

- Tasks are composed of
  → code to execute, a data environment (initialized at creation time), internal control variables (ICVs)

- Tasks are created…
  … when reaching a parallel region → implicit tasks are created (per thread)
  … when encountering a task construct → explicit task is created
  … when encountering a taskloop construct → explicit tasks per chunk are created
  … when encountering a target construct → target task is created
OpenMP Tasking Idiom

- OpenMP programmers need a specific idiom to kick off task-parallel execution: `parallel masked`
  - OpenMP version 5.0 introduced the `parallel master` construct
  - With OpenMP version 5.1 this becomes `parallel masked`

```c
int main(int argc, char* argv[]) {
    [...] #pragma omp parallel
    {
        #pragma omp masked
        {
            start_task_parallel_execution();
        }
    }
    [...] start_task_parallel_execution();
}
```

```c
int main(int argc, char* argv[]) {
    [...] #pragma omp parallel
    {
        #pragma omp single
        {
            start_task_parallel_execution();
        }
    }
    [...] start_task_parallel_execution();
}
```
Fibonacci Numbers (in a Stupid Way 😊)

- Only one thread enters `fib()` from `main()`.
- That thread creates the two initial work tasks and starts the parallel recursion.
- The `taskwait` construct is required to wait for the result for `x` and `y` before the task can sum up.
- T1 enters fib(4)
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 enters `fib(4)`
- T1 creates tasks for `fib(3)` and `fib(2)`
- T1 and T2 execute tasks from the queue
- T1 enters `fib(4)`
- T1 creates tasks for `fib(3)` and `fib(2)`
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- T1 enters fib(4)
- T1 creates tasks for fib(3) and fib(2)
- T1 and T2 execute tasks from the queue
- T1 and T2 create 4 new tasks
- T1 - T4 execute tasks
- ...
Programming OpenMP

Hands-on Exercises

Christian Terboven
Michael Klemm
Exercises

- We have implemented a series of small hands-on examples that you can use and play with.
  - Download: https://github.com/NERSC/openmp-series-2024
  - Build: make

- Each hands-on exercise has a folder “solution”
  - It shows the OpenMP solution that we have added
  - You can use it to cheat 😊, or to check if you came up with the same solution
Exercises: Overview

<table>
<thead>
<tr>
<th>Exercise no.</th>
<th>Exercise name</th>
<th>OpenMP Topic</th>
<th>Day / Order (proposal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hello World</td>
<td>Getting started</td>
<td>Start with this (if OpenMP is new for you)</td>
</tr>
<tr>
<td>2</td>
<td>Pi</td>
<td>Worksharing, Scoping</td>
<td>First day</td>
</tr>
<tr>
<td>3</td>
<td>Jacobi</td>
<td>Worksharing, Scoping</td>
<td>First day</td>
</tr>
<tr>
<td>4</td>
<td>Work-Distribution</td>
<td>Worksharing</td>
<td>First day</td>
</tr>
<tr>
<td>5</td>
<td>Min/Max</td>
<td>Worksharing, Reduction</td>
<td>First day</td>
</tr>
</tbody>
</table>
to be continued …